

Versatile High Performance Hierarchies in SQL Server

Page 1 of 3 - by [Dennis W. Forbes](#)

New! <http://www.professionalsqlserver.com>. The premiere source of SQL Server tips, tricks and information

[The #1 Internet Resource For SQL Server Professionals](#)

Introduction



Figure 1 : Inverted Tree

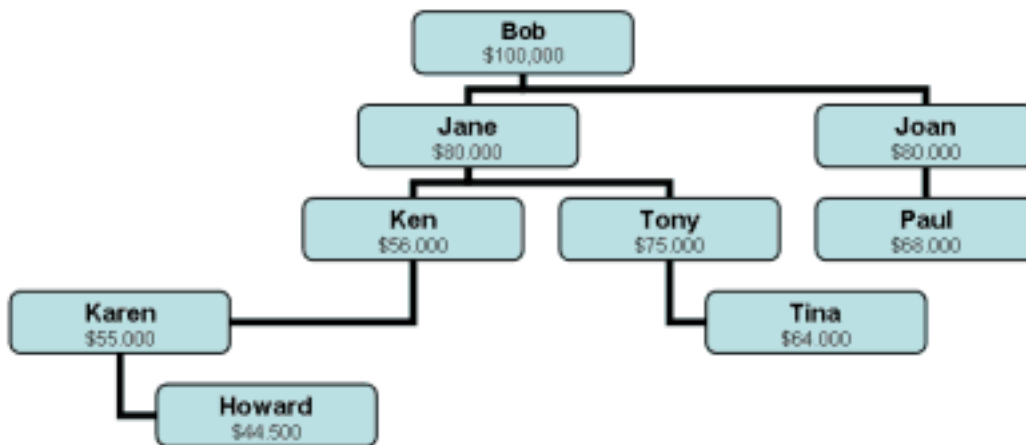


Figure 2 : Corporate Reporting Hierarchy

EmployeeID FullName Boss1ID Boss2ID Boss3ID Boss4ID

1	Bob	Null	Null	Null	Null
2	Jane	1	Null	Null	Null
3	Joan	1	Null	Null	Null
4	Ken	2	1	Null	Null
5	Tony	2	1	Null	Null
6	Karen	4	2	1	Null
7	Paul	3	1	Null	Null
8	Howard	6	4	2	1
9	Tina	5	2	1	Null

Figure 3 : Multi-Referential Employee Table

EmployeeID FullName BossID

1	Bob	Null
2	Jane	1
3	Joan	1
4	Ken	2
5	Tony	2
6	Karen	4
7	Paul	3
8	Howard	6
9	Tina	5

Figure 4 : Adjacency List

```

CREATE FUNCTION dbo.GetReports(@IncludeParent bit, @EmployeeID int)
RETURNS @retFindReports TABLE (EmployeeID int, Name varchar(50), BossID int)
AS
BEGIN
    IF (@IncludeParent=1)
    BEGIN
        INSERT INTO @retFindReports
        SELECT * FROM Employees WHERE EmployeeID=@EmployeeID
    END
END

```

```

DECLARE @Report_ID int, @Report_Name varchar(50), @Report_BossID int

DECLARE RetrieveReports CURSOR STATIC LOCAL FOR
SELECT * FROM Employees WHERE BossID=@EmployeeID

OPEN RetrieveReports

FETCH NEXT FROM RetrieveReports
INTO @Report_ID, @Report_Name, @Report_BossID

WHILE (@@FETCH_STATUS = 0)
BEGIN
    INSERT INTO @retFindReports
    SELECT * FROM dbo.GetReports(0,@Report_ID)

    INSERT INTO @retFindReports
    VALUES(@Report_ID,@Report_Name, @Report_BossID)

    FETCH NEXT FROM RetrieveReports
    INTO @Report_ID, @Report_Name, @Report_BossID
END

CLOSE RetrieveReports
DEALLOCATE RetrieveReports

RETURN
END

```

Figure 5 : GetReports User Defined Function

A commonly modeled database design is the ubiquitous hierarchy: A relationship of items, all but the root having a single “parent”, with each having zero or more child nodes. Such a structure is often referred to as an “inverted tree” due to its visualized similarity to, not surprisingly, an inverted tree ([Figure 1](#)). Examples of such hierarchies include corporate reporting structures ([Figure 2](#)), geographical groupings, inventory tracking (such as when parts are assembled into components, which themselves might be combined into products, which might then become combined into shipping units, etc), auction listing categories, etc. The file system on our computers, a tree of folders and files, is a common example of such a hierarchical structure.

Unfortunately, hierarchies present a quandary when implemented within a data model: The most versatile, highly-relational designs can be complex and inefficient to query upon, while the more limited, hard-coded solutions offer short-term benefits, yet turn into a critical weakness as systems expand. This document gives some techniques to achieve the best of both worlds: A powerful method of creating simple, self-maintaining, versatile, relational hierarchical sets.

Hierarchical Data Models

There are several methods of implementing a hierarchy in a database. One common approach is to

include references to each successively higher level in each record, with all of the levels sharing a single table. An example of this multi-level relational design in common use is the standard address: street, city, state, and country – All levels are often stored with each address entry, rather than having the record reference a street record, which itself links to a city record, and so on. In a corporate reporting structure such a solution could be implemented via a link in each person's record to their supervisor, their supervisor's supervisor, and so on ([Figure 3](#)), to the depth required given the vertical height of the organization's organizational chart. Such a technique allows for generally straightforward querying, such as the following query which computes the total salary of all of Jane's reports.

```
SELECT Sum(Salary) FROM Employees WHERE Boss1ID=2 OR Boss2ID=2 OR Boss3ID=2 or Boss4ID=2
```

Such a structure is easy to query upon, however it has a large degree of redundancy--a change of hierarchical relationships can cascade across rows and columns in a complex manner-- and lacks flexibility--If additional levels were added it would require a schema change (i.e. the addition of columns). This isn't appropriate for anything but the most simplistic and static of hierarchies.

[Figure 4](#) demonstrates another, simplified method of storing a hierarchy. This technique, often called an *adjacency list*, is a variation of the structure described in [Figure 3](#), but without the redundant information-- Only the parent ID is stored, as successively higher levels can be determined by walking up the hierarchy programmatically. Such an approach offers versatility, and is an efficient choice for irregular (jagged) trees that go deep on some branches while remaining shallow on others, as the depth of the tree is not hard coded into the schema design. Whole branches can be moved by changing a parent relationship in a single record, and there is minimal space requirements imposed by the parent field used to create the relationships. However, adjacency lists can be complex to query in a hierarchical fashion without loads of self-joins or iterative loops: Answering a question such as "What is the total salary of Jane's division?" can be a complex task, requiring the construction of temporary tables and costly recursive calls. Previously such a programmatic solution was difficult to implement in practice within SQL Server, however the task is made easier using the inline table-valued User Defined Functions functionality of SQL Server 2000.

[Figure 5](#) demonstrates a user defined function (UDF) which takes two parameters: The first parameter, *@IncludeParent*, is a boolean value indicating whether to include the parent record in the result set, while the second parameter, *@EmployeeID*, is an integer identifying the employee whose subordinates we desire. This function returns a table containing all descendant records. This result set can be used for aggregate functions, such as the following query that returns the sum of the salaries for all reports under Jane's direct or indirect command, excluding Jane herself.

```
SELECT sum(Salary) FROM dbo.GetReports(0,2)
```

Another common task would be to perform searches in branches of the tree. For example "I remember meeting a guy named Howard in Jane's division...who was that guy?"

```
SELECT * FROM dbo.GetReports(0,2) WHERE FullName='Howard'
```

This function could be also used with the powerful *IN* condition for modifiable resultsets.

```
SELECT * FROM Employees WHERE EmployeeID IN (SELECT EmployeeID FROM dbo.GetReports(0,2))
```

SQL Server 2005 Update (2004-08-08)

SQL Server 2005, formerly codenamed Yukon, offers a simple T-SQL native mechanism for querying recursive sets - Recursive Common Table Expressions (Recursive CTE). CTEs could be considered a type of dynamic inline view, with the added benefit that they also offer self-referential functionality. Presuming that the adjacency table in [Figure 4](#) were called "Employees", the following T-SQL would pull all employees reporting directly or indirectly to Jane.

```
DECLARE @boss_id int
```

```
SET @boss_id = 2;
```

```
WITH CTE_Example (EmployeeID, FullName, BossID, Depth)
```

```
AS
```

```
(
```

```
    SELECT EmployeeID, FullName, BossID, 0 AS Depth
```

```
    FROM Employees WHERE EmployeeID = @boss_id
```

```
    UNION ALL
```

```
    SELECT Employees.EmployeeID, Employees.FullName, Employees.BossID, CTE_Example.Depth + 1 AS Depth
```

```
FROM Employees
```

```
    JOIN CTE_Example ON Employees.BossID = CTE_Example.EmployeeID
```

```
)
```

```
SELECT * FROM CTE_Example
```

In no way does Recursive CTE functionality invalidate the string nested set technique: Under the covers a recursive CTE is simply some syntactical sugar that functionally performs largely the same operation as seen in [Figure 5](#).

[Go To Page 2 - High Performance Nested Sets](#)